50.007 Machine Learning
Project Report

1006236	Shelen Go
1006242	Michelle Ordelia Sumaryo
1006365	Ruan Yang
1006071	Rachel Leow Si Min

PART 1

Section 1

```python
from fractions import Fraction

#function that estimates the emission parameters from the training set using MLE
def estimate_emission_parameters_mle(training_data):
    emission_params = {}    # emission parameters initialization
    emission_counts = {}    # initialize emission counts of each word given each tag

    # Looping through the training data
    for sentence in training_data:
        for word, tag in sentence:
            if tag not in emission_counts:
                emission_counts[tag] = {}
            if word not in emission_counts[tag]:
                emission_counts[tag][word] = 0
            emission_counts[tag][word] += 1

    # Calculating MLE
    for tag, word_counts in emission_counts.items():
        total_count = sum(word_counts.values())
        # sub-dictionary for current tag in emission_params
        emission_params[tag] = {}
        for word, count in word_counts.items():
            emission_params[tag][word] = count / total_count
            # emission_params[tag][word] = Fraction(count, total_count)

    return emission_params
```

We call the `estimate_emission_parameters_mle` to estimate the emission probabilities within a Hidden Markov Model (HMM) using Maximum Likelihood Estimation (MLE). The input to this function is the train dataset. Firstly, initialize the dictionaries to store emission probabilities and counts of words associated with each tag. We then iterate through the training data, incrementing the emission counts for each word-tag pair. Subsequently, it calculates the emission probabilities for each tag-word combination by dividing the count of a specific word under a certain tag by the total count of all words under that tag. It outputs a dictionary containing the estimated emission probabilities, which indicate the likelihood of observing particular words given specific tags.

Section 2

```python
def estimate_emissions_from_data(data, k=1):
    sentiment_count = {}
    emission_parameters = {}

    for sentence in data:
        for word, sentiment in sentence:
            sentiment_count.setdefault(sentiment, 0)
            sentiment_count[sentiment] += 1

            emission_parameters.setdefault(word, {}).setdefault(sentiment, 0)
            emission_parameters[word][sentiment] += 1

    emission_parameters["#UNK#"] = {}
    for sentiment in sentiment_count:
        emission_parameters['#UNK#'][sentiment] = k

    for word, sentiment_count_dict in emission_parameters.items():
        for sentiment, count in sentiment_count_dict.items():
            emission_parameters[word][sentiment] = count / sentiment_count[sentiment]

    return emission_parameters
```
✓ 0.0s Python

We call the function `estimate_emissions_from_data` which uses 2 dictionaries, sentiment_count and emission_parameters to store word-sentiment counts. We iterated through the data and apply additive smoothing with a parameter k. The function calculates normalized emission probabilities.

Section 3

```python
def read_dataset(file_path, labeled=True):
    sentences = []          # Holds all the sentences parsed from the file
    current_sentence = []   # Holds the words and tags of current sentence being processed

    with open(file_path, 'r', encoding='utf-8') as f:
        for line_number, line in enumerate(f, start=1):
            if line == "" or line == '' or line == "\n":
                if current_sentence:
                    sentences.append(current_sentence)
                current_sentence = []
            else:
                line = line.strip() # Remove leading/trailing whitespaces

                # Check if the line is not empty
                if line:
                    if labeled:
                        tokens = line.split()
                        if len(tokens) >= 2:
                            word, tag = ' '.join(tokens[:-1]), tokens[-1] # join parts except the las
                            current_sentence.append((word, tag))
                        else:
                            print(f"Error in line {line_number}: Unexpected format - {line}")
                    else:
                        current_sentence.append(line)

    # If there's remaining sentence at the end, add it to the list of sentences
    if current_sentence:
        sentences.append(current_sentence)

    return sentences
```

```python
def UNK(dev,words):
    dev_2 = copy.deepcopy(dev)
    for sentence in dev_2:
            for index, word in enumerate(sentence):
                if word not in words:
                    sentence[index] = "#UNK#"
    return dev_2
```

✓ 0.0s

```python
def simple_sentiment_analysis(dataset, trainset, output_path):
    words = set([x for sentence in trainset for (x, y) in sentence])
    dataset = UNK(dataset, words)
    emission_params = estimate_emissions_from_data(trainset)
    output = []
    for sentence in dataset:
        sentence_wlabels = []
        for word in sentence:
            sentiment_dict = emission_params.get(word, {})
            if sentiment_dict:
                label = max(sentiment_dict, key=sentiment_dict.get)
                sentence_wlabels.append((word, label))
            else:
                sentence_wlabels.append((word, 'O'))
        output.append(sentence_wlabels)
    write_file(output, output_path)
```

```python
ES_dev_in = read_dataset('./ES/dev.in', labeled=False)
ES_data_train = read_data('./ES/train')
simple_sentiment_analysis(ES_dev_in ,ES_data_train, "./ES/dev.p1.out")
```

✓ 0.1s

```python
RU_dev_in = read_dataset('./RU/dev.in', labeled=False)
RU_data_train = read_data('./RU/train')
simple_sentiment_analysis(RU_dev_in ,RU_data_train, "./RU/dev.p1.out")
```

✓ 0.1s

Our code defines functions to perform simple sentiment analysis on a dataset. The `read_dataset` function reads sentences from a file, tokenizing and tagging them with words and labels. The `UNK` function replaces out-of-vocabulary words in the given dataset with a special "#UNK#" token. The `estimate_emissions_from_data` function calculates emission probabilities for words and sentiment labels based on training data. The `simple_sentiment_analysis` function takes an input dataset, replaces unknown words with "#UNK#", estimates emission probabilities using the training set, and assigns sentiment labels to words based on the highest emission probability. The results are then written to an output file. This process enables basic sentiment analysis by associating sentiment labels with words in a given dataset. Subsequently, the code writes these predictions to an output file named dev.p1.out.

Results

Results of precision, recall and F scores of such a baseline system for each dataset:

```
shelengo@Shelens-MacBook-Pro ES % python3 evalResult.py dev.out dev.p1.out

#Entity in gold data: 229
#Entity in prediction: 1466

#Correct Entity : 178
Entity  precision: 0.1214
Entity  recall: 0.7773
Entity  F: 0.2100

#Correct Sentiment : 97
Sentiment  precision: 0.0662
Sentiment  recall: 0.4236
Sentiment  F: 0.1145
```

```
shelengo@Shelens-MacBook-Pro RU % python3 evalResult.py dev.out dev.p1.out

#Entity in gold data: 389
#Entity in prediction: 1816

#Correct Entity : 266
Entity  precision: 0.1465
Entity  recall: 0.6838
Entity  F: 0.2413

#Correct Sentiment : 129
Sentiment  precision: 0.0710
Sentiment  recall: 0.3316
Sentiment  F: 0.1170
```

PART 2

Section 1

```python
import sys
sys.stdout.reconfigure(encoding='utf-8')
with open("./Data/ES/train", "r", encoding="utf-8") as f:
    data = f.read()

# CALCULATE: TRANSITION PARAMETER
## q(yi|yi-1) = Count(yi-1, yi) / Count(yi-1)

def transitionParameter(data, yi_1) :

    lines = data.split('\n')
    transition_probabilities = {}

    ## CALCULATE: Count(yi-1, yi)
    transition_counts = {}
    for i in range(1, len(lines)):
        yi_1, yi = lines[i - 1], lines[i]

        # Updating Counts
        if (yi_1, yi) in transition_counts:
            transition_counts[(yi_1, yi)] += 1
        else:
            transition_counts[(yi_1, yi)] = 1

    ## CALCULATE: Count(yi-1)
    for pair, count in transition_counts.items():
        yi_1, yi = pair
        count_yi_1 = sum(1 for line in lines if line.startswith(yi_1))  # Count(yi-1)

        # CALCULATE: q(yi | yi-1) = Count(yi-1, yi) / Count(yi-1)
        transition_probabilities[pair] = count / count_yi_1

    return transition_probabilities
```

We call the function which takes in 2 argument data and the previous state. The function initializes an empty dictionary which stores the calculated probabilities. Firstly, the function counts the transition count. The code parses the data by splitting it into lines and then sequentially processes these lines. Starting from the second line (index 1), the function extracts the current state (yi) and the previous state (yi_1) from each line. We track the transition from y_I to yi in the transition_counts. For every transition, the function updates the corresponding entry in the transition_counts by either incrementing an existing count or initializing it to 1, in the base case. Hence the function can then determine the transition probabilities for each pair using the given formula. The resultant probabilities are stored within the transition_probabilities dictionary.

Section 2

```python
max_score = []
best_route = []
backtrack_dict = {}
backtrack_list = []
best_score= {}
possible = ["B-positive", "I-positive", "B-neutral", "I-neutral", "O", "B-negative", "I-negative"]
punctuations = [".", ",", "!", '"', "'", "?", "(", ")", ":", "..."]
counter=[]

# init viterbi class
viterbi = viterbi_algorithm()

for sentence in prepped_test_data:
    viterbi.run(transition_params, emission_params, sentence)
```

This is the code to run the Part2 Section2 Code.

The run method initializes the necessary parameters and calls the viterbi_main method, which is the core of the algorithm. Within the viterbi_main method, the algorithm starts with a base case for the start of the sequence, setting the initial score to 1. For each subsequent step, the algorithm examines the current word and calculates the best score for transitioning to different possible states based on the transition and emission probabilities. Looking at the given dataset (ES/RU), special handling is required for punctuation and digital characters, which are assumed to be in the “O” category. The algorithm iterates through the possible tags and computes scores based on transition and emission probabilities. Unknown words are addressed through an “#UNK#” category. Once the highest-scoring state for the current time step is determined, the sequence of best tags is constructed incrementally. The algorithm iterates recursively backward and then frontwards through the sequence. The final prediction is based on the best route of tags found through the algorithm.

```python
class viterbi_algorithm:
    def run(self, transition, emission, data):
        j = len(data)
        k = len(data)
        self.u = "START"
        self.m = "STOP"
        forward_result_score = self.viterbi_main(j, transition, emission, data, k, data)
        return forward_result_score

    def viterbi_main(self, j, transition, emission, data, k, sentence):
        #base case, start
        if j == 0:
            max_score.append(1)
            return 1
        else:
            print(j)
            current = data[j - 1]
            # managed to pass each word in recursion step
            result = self.viterbi_main(j - 1, transition, emission, data[0 : j-1], k, sentence)
            print(self.u)
            # print(data[j - 1])

            if j <= k:
                #prior knowledge, puctuations are O
                if current in punctuations:
                    score = result * transition[self.u, "O"] * 1
                    self.u = "O"
                    best_route.append("O")
                    return score
                else:
                    for char in current:
                        if char.isdigit():
                            score = result * transition[self.u, "O"] * 1
                            self.u = "O"
                            best_route.append("O")
                            return score
                # think about when j -1 = 1, first word "Plato"
                temp_dict = {}

                #finding best v score of all possible states of v
                for v in possible:
                    # print(result)
```

```python
                #finding best v score of all possible states of v
                for v in possible:
                    # print(result)
                    try:
                        score = result * transition[self.u, v] * emission[current][v]
                    # not found in the parameters, means no occurence
                    except KeyError:
                        try:
                            score = result * transition[self.u, v] * emission["#UNK#"][v]
                        except KeyError:
                            score = 0
                    best_score.update({str(j)+","+v: score})
                    temp_dict.update({v: score})

                    # print(temp_dict)

                max_value = max(temp_dict.values())
                max_keys = [key for key, value in temp_dict.items() if value == max_value]
                final_key = max_keys[0]

                self.u = final_key
                # print(f"{max_keys}: {max_value}\n")
                best_route.append(self.u)
                return max_value
```

Results

```
#Entity in gold data: 229
#Entity in prediction: 642

#Correct Entity : 111
Entity  precision: 0.1729
Entity  recall: 0.4847
Entity  F: 0.2549

#Correct Sentiment : 76
Sentiment  precision: 0.1184
Sentiment  recall: 0.3319
Sentiment  F: 0.1745
```

ES DATA

```
#Entity in gold data: 389
#Entity in prediction: 381

#Correct Entity : 138
Entity  precision: 0.3622
Entity  recall: 0.3548
Entity  F: 0.3584

#Correct Sentiment : 94
Sentiment  precision: 0.2467
Sentiment  recall: 0.2416
Sentiment  F: 0.2442
```

RU DATA

PART 3

```python
from itertools import product
with open("./Data/RU/train", "r", encoding="utf-8") as f:
    data = f.read()

transition_params = transitionParameter(data)
# print(transition_params)

prepped_test_data = prepare_data('RU')
# print(len(prepped_test_data))

test_results = getting_tag("RU")

with open('./Data/RU/train', "rb") as f:
    train_data = f.read()

passInto = [line.decode('utf-8') for line in train_data.split(b'\n')]
# print(passInto)
emission_params = estimate_emissions(passInto)

class kth_viterbi_algorithm:
    def run(self, transition, emission, data):
        j = len(data)
        k = len(data)
        self.u = "START"
        self.m = "STOP"
        forward_result_score = self.modified_viterbi_main(j, transition, emission, data, k, data)
        result = modified_viterbi.get_kth_path(2)
        kth_store = []
        return forward_result_score

    def modified_viterbi_main(self, j, transition, emission, data, k, sentence):
        #base case, start
        if j == 0:
            max_score.append(1)
```

```python
        else:
            # print(j)
            current = data[j - 1]
            # managed to pass each word in recursion step
            result = self.modified_viterbi_main(j - 1, transition, emission, data[0 : j-1], k, sentence)
            # print(self.u)
            # print(data[j - 1])

            if j <= k:
                #prior knowledge, puctuations are O
                # if current in punctuations:
                #     score = result * transition[self.u, "O"] * 1
                #     self.u = "O"
                #     best_route.append("O")
                #     return score
                # else:
                #     for char in current:
                #         if char.isdigit():
                #             score = result * transition[self.u, "O"] * 1
                #             self.u = "O"
                #             best_route.append("O")
                #             return score
                # think about when j -1 = 1, first word "Plato"
                temp_dict = {}

                if j+1 == k:
                    self.result = result
                    kth_tracking = {}
                    for i in possible:
                        # print("hi")
                        for v in possible:
                            try:
                                score = transition[i, v] * emission[current][v]
                            # not found in the parameters, means no occurence
                            except KeyError:
```

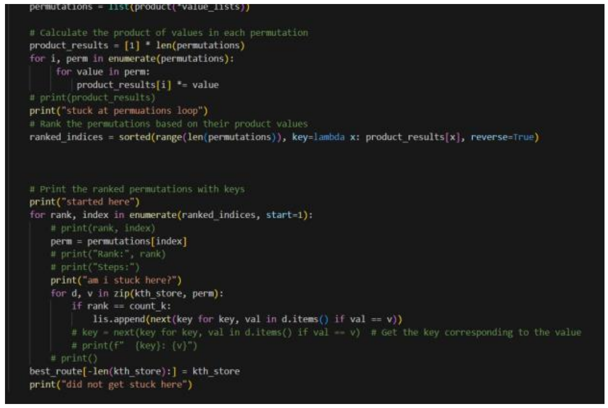
```python
                #finding best v score of all possible states of v
                print("im not stuck")
                for v in possible:
                    # print(result)
                    try:
                        score = result * transition[self.u, v] * emission[current][v]
                    # not found in the parameters, means no occurence
                    except KeyError:
                        try:
                            score = result * transition[self.u, v] * emission["#UNK#"][v]
                        except KeyError:
                            score = 0
                    best_score.update({str(j)+","+v: score})
                    temp_dict.update({v: score})
                    # print(temp_dict)

                max_value = max(temp_dict.values())
                max_keys = [key for key, value in temp_dict.items() if value == max_value]
                final_key = max_keys[0]

                self.u = final_key
                # print(f"{max_keys}: {max_value}\n")
                best_route.append(self.u)
                print("finished one")
                return max_value

    def get_kth_path(self, count_k):

        # Extract values from each dictionary in the list
        # print(len(kth_store))
        value_lists = [list(d.values()) for d in kth_store]
        # print(len(value_lists))
        # Generate permutations by selecting one value from each dictionary
        permutations = list(product(*value_lists))
```



```
permutations = list(product(*value_lists))

# Calculate the product of values in each permutation
product_results = [1] * len(permutations)
for i, perm in enumerate(permutations):
    for value in perm:
        product_results[i] *= value
# print(product_results)
print("stuck at permuations loop")
# Rank the permutations based on their product values
ranked_indices = sorted(range(len(permutations)), key=lambda x: product_results[x], reverse=True)



# Print the ranked permutations with keys
print("started here")
for rank, index in enumerate(ranked_indices, start=1):
    # print(rank, index)
    perm = permutations[index]
    # print("Rank:", rank)
    # print("Steps:")
    print("am i stuck here?")
    for d, v in zip(kth_store, perm):
        if rank == count_k:
            lis.append(next(key for key, val in d.items() if val == v))
        # key = next(key for key, val in d.items() if val == v)  # Get the key corresponding to the value
        # print(f"  {key}: {v}")
    # print()
best_route[-len(kth_store):] = kth_store
print("did not get stuck here")
```

We implemented a structured prediction problem where we implemented a modified version of our Viterbi algorithm to find the k-best paths for a sequence of observations.

1. Preparing for K-Best Paths:
    a. The kth_viterbi_algorithm is an extension of the viterbi algorithm to handle the k-best paths. It accumulates k-best paths in kth_store and keeps track of various intermediate data structures.
2. Viterbi Algorithm Execution:
    a. The Viterbi algorithm is excuted for each sentence in prepped_test_data using the modified_viterbi_algorithm method. We find the possible paths for each sentence.
3. K-Best Paths Calculation:
    a. After executing the Viterbi algorithm for each sentence, the get_kth_path method od kth_viterbi_algorithm is called with a specific value of k(2 in our case) to retrieve the k-best paths for each sentence. This method is responsible for populating the kth_store list with dictionaries containing probabilities or scores for each state in the path.
4. Extracting Values and Calculating Permutations
    a. We then extract the values from the kth_store list of dictionaries, generating permutations of those values, and calculating products of values in each permutations. We will rank and select the k-best paths based on the calculated product values.
5. Finalizing and Writing Output
    a. After the k-best paths are calculated and stored in k-th store the code reads the contents of a file line by line and assigns the corresponding tags based on the best route or k-best paths obtained. The modified lines are then written back to another file (dev.p3.out)

PART 4

```python
import sys
sys.stdout.reconfigure(encoding='utf-8')
with open("./Data/ES/train", "r", encoding="utf-8") as f:
    data = f.read()

## INPUT: list of lines
## OUTPUT: List of word-tags from data

def word_and_tag(data):
    lines = data.split('\n')
    words_tags = []

    for line in lines:
        elements = line.strip().split()
        if len(elements) == 2:
            word, tag = elements
            words_tags.append((word, tag))
        else:
            words_tags.append(("", ""))
    return words_tags

class viterbi_algorithm:
    def run(self, transition, emission, data):

        state_scores = {}

        j = len(data)
        k = len(data)
        self.u = "START"
        self.m = "STOP"
        forward_result_score = self.viterbi_main(j, transition, emission, data, k, data, {})
        return forward_result_score, state_scores

    def viterbi_main(self, j, transition, emission, data, k, sentence, state_scores):
        #base case, start
        if j == 0:
            max_score.append(1)
```

```python
        if j == 0:
            max_score.append(1)
            return 1
        else:
            print(j)
            current = data[j - 1]
            # managed to pass each word in recursion step
            result = self.viterbi_main(j - 1, transition, emission, data[0 : j-1], k, sentence, state_scores)
            print(self.u)
            # print(data[j - 1])

            if j <= k:
                #prior knowledge, puctuations are O
                if current in punctuations:
                    score = result * transition[self.u, "O"] * 1
                    self.u = "O"
                    best_route.append("O")
                    return score
                else:
                    for char in current:
                        if char.isdigit():
                            score = result * transition[self.u, "O"] * 1
                            self.u = "O"
                            best_route.append("O")
                            return score

                temp_dict = {}

                #finding best v score of all possible states of v
                for v in possible:
                    try:
                        score = result * transition[self.u, v] * emission[current][v]
                    except KeyError:
                        try:
                            score = result * transition[self.u, v] * emission["#UNK#"][v]
                        except KeyError:
                            score = 0
```

```python
                    state_scores.update({v: score})  # Update state_scores dictionary

                    temp_dict[v] = score

                max_value = max(temp_dict.values())
                max_keys = [key for key, value in temp_dict.items() if value == max_value]
                final_key = max_keys[0]

                self.u = final_key
                best_route.append(self.u)
                return max_value, state_scores


#calculation = np.log(emission_matrix) + np.log(transition_matrix)

## IMPLEMENTING STRUCTURED PERCEPTRON


def teach_perceptron(word_tag_data, possible, num_epochs, emission_probs, transition_probs):

    # Initialise WEIGHT DICTIONARY --> each tag as a key and an initial weight of 0
    weights = {label: 0 for label in possible if label != "START"}
    learn_rate = 0.1
    transition_counts = {}

    for _ in range(num_epochs):
        for i, (term, tag) in enumerate(word_tag_data):

            ## START OF NEW SENTENCE
            if word_tag_data[i-1][0] == "":
                current_term = term
                j = i
                while current_term != "":
                    j += 1
                    current_state = tag
                    next_state = word_tag_data[j][1]
                    if current_state not in transition_counts:
```

```python
                while current_term != "":
                    j += 1
                    current_state = tag
                    next_state = word_tag_data[j][1]
                    if current_state not in transition_counts:
                        transition_counts[current_state] = {next_state: 1}
                    elif next_state in transition_counts[current_state]:
                        transition_counts[current_state][next_state] += 1
                    current_term = word_tag_data[j][0]
                current_term = ''
                j = 0

            ## IMPROVING VITERBI
            # Predict the best tag (state) for the current word (term)
            if term:
                state_scored = viterbi_algorithm.run(transition_probs, emission_probs, [term])
                predicted_state = max(state_scored, key=state_scored.get)

                ## MISCLASSIFICATION:
                if predicted_state != tag:
                    next_term, next_state = word_tag_data[i+1]
                    real_transition = transition_probs.get(next_state, {}).get(tag, 0)
                    real_counts = transition_counts.get(tag, {}).get(next_state, 0)
                    pred_counts = transition_counts.get(tag, {}).get(predicted_state, 0)

                    weights[tag] = learn_rate * (weights.get(tag) + (real_transition * real_counts))
                    weights[predicted_state] = learn_rate * (weights.get(predicted_state) - state_scored[predicted_state]

    return weights
```

The structured perceptron algorithm enhances the Viterbi algorithm as it iteratively refines the emission and transition probabilities used for sequence tagging. While the Viterbi algorithm focuses on finding the most likely sequence of tags based solely on given probabilities, the structured perceptron incorporates a training process. It updates the probabilities during each iteration to minimize classification errors, effectively adapting the model to the training data.

The structured perceptron achieves this through the following steps:

1. It initially assigns weights to each tag, representing their importance in the tagging decision.
2. During training iterations, it employs the Viterbi algorithm to predict tags for each word and then compares these predictions to the actual tags in the training data.
3. When a misclassification occurs, the algorithm adjusts the weights associated with the true and predicted tags based on their probabilities and counts.
4. This weight adjustment mechanism optimizes the model's performance by encouraging the correct tags and discouraging incorrect ones.

By iteratively refining the model's parameters, the structured perceptron improves the accuracy of sequence tagging.